

# DOCUMENTATION TECHNIQUE DE NOTRE PROJET UNITY

## Introduction :

Avant de présenter plus en détail les notions techniques du projet, nous allons présenter son contexte.

Le projet est un jeu style TPS où le joueur devra trouver la zone d'extraction et sortir du hangar infesté de zombies. Des kits de soins et de munitions lui viendront en aide dans la map. Attention, vous avez 5 minutes ! Si vous y parvenez, vous intégrerez les rangs du S.W.A.T. Bonne chance, soldat !

## Fonctionnalités du jeu :

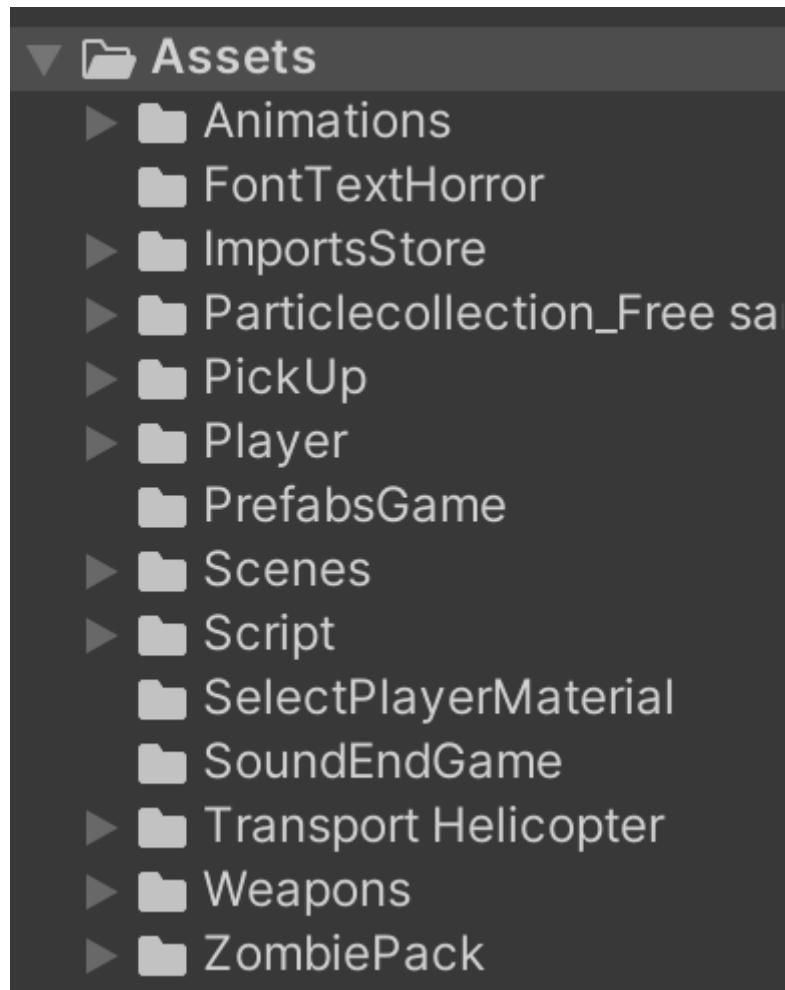
Voici tout ce que nous avons fait dans le jeu ou ce qui est possible de faire.

- Un menu principal (play/quit)
- Un menu sélection du joueur qui permet de choisir entre 2 skins
- Une map de jeu principal avec :
  - Une safe zone :
    - Apparition du joueur sélectionné, il peut :
      - sauter (distinction de sur place ou en courant avec espace)
      - direction (Z Q S D)
      - s'accroupir (Ctrl gauche et se déplacer accroupi)
      - marcher
      - courir (Shift gauche)
      - viser (clique droit)
      - tirer (clique gauche)
      - recharger (automatiquement à 0 ou en appuyant sur R + son)
      - utiliser un kit de vie (en appuyant sur H et vie < max vie)
      - utiliser des kits de munition (se déplacer dessus quand munitions < munitions max)
    - Un pnj avec qui l'on peut interagir (E) et qui nous donne quelques instructions.
    - Un son d'ambiance dans la map.
    - Un kit de soin et munition que l'on peut ramasser et utiliser + son ramassage
    - 3 portes qui s'ouvrent lorsque l'on se présentent devant elles
    - 3 boucliers qui déclenchent le chrono lorsque l'on passe à travers la première fois.

- Le hangar :
  - On y retrouve des kits de vie et de munitions
  - Des zombies type A, B et C avec des skins différents et des caractéristiques différentes qui pourchasse le joueur.
  - Ces zombies spawns à des endroits différents, à des intervalles différents et en nombre différents (surpriiiiise !)
  - Le joueur perd des points de vie si :
    - Il rentre en contact avec un zombie (le touche)
    - un zombie l'attaque
    - il saute de trop haut
  - Si le joueur perd des points de vie, du sang s'affiche autour de l'écran puis disparaît au bout de 2s, sauf si ses points de vie sont < 30, tant qu'il n'utilise pas de kits de soins cela restera.
  - Si le joueur meurt :
    - le joueur ne peut plus rien faire
    - le timer s'arrête
    - Un menu de mort s'affiche (replay, main menu, quit)
  - une zone d'extraction est à trouver pour s'échapper du hangar
  - une fois cette zone atteinte :
    - le timer s'arrête
    - le personnage s'élève (vous ne pouvez plus rien faire, profitez de la fin du jeu)
    - une fois en hauteur et dehors :
      - le menu de fin s'affiche (replay, main menu, quit)
      - vous verrez un avion de transport animé vous attendre et un pnj du swat pour vous extraire
      - des zombies danser sur le toit pour célébrer votre victoire

## **Architecture du projet :**

Voici comment se présente l'architecture du projet :



Nous avons mis en place le modèle “MVC” où les vues sont rangées dans le dossier Scenes, les modèles et controllers dans PrefabsGame et Script. Nous avons également des dossiers pour chaque éléments différents du jeu afin de s'y retrouver plus facilement.

Notre jeu se compose de 3 scenes :

- Une pour le menu principal.
- Une autre pour la sélection du joueur.
- La dernière qui constitue la map du jeu.

## Menu principal :



Pour réaliser cette scène, nous avons importé notre map générée à partir d'assets que nous avons utilisé.

La "main" caméra a été placée à un endroit qui nous semblait adapté puis nous avons construit un canvas qui contient les éléments du menu principal :

- Un texte : Menu.
- Un button Play.
- Un button Quit.
- Un texte : HACED (le nom de notre jeu).
- 2 images contenant des sprites de zombies.
- 2 images contenant des sprites d'impacts de balles en sang.

Un script qui permet d'effectuer les actions adéquates lorsque nous cliquons sur les boutons a été créé. Celui-ci est stocké dans l'objet : MainMenuManager et le script porte le même nom.

```
public class MainMenuManager : MonoBehaviour
{
    [SerializeField] private GameObject mainMenu;

    private void Start() => ActivateMainMenu(true);

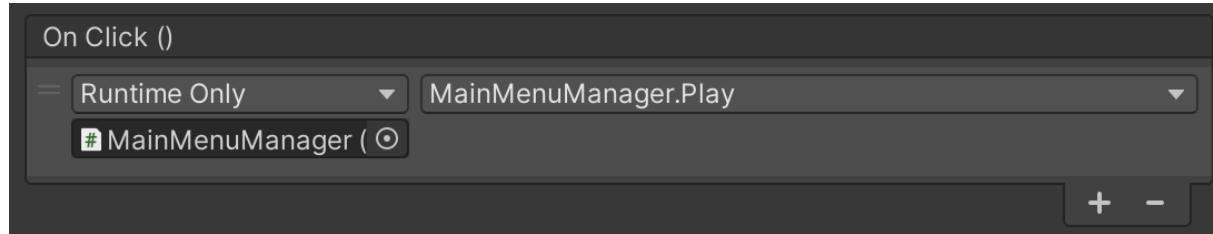
    public void ActivateMainMenu(bool state) => mainMenu.SetActive(state);

    public void Play() => SceneManager.LoadScene(1);

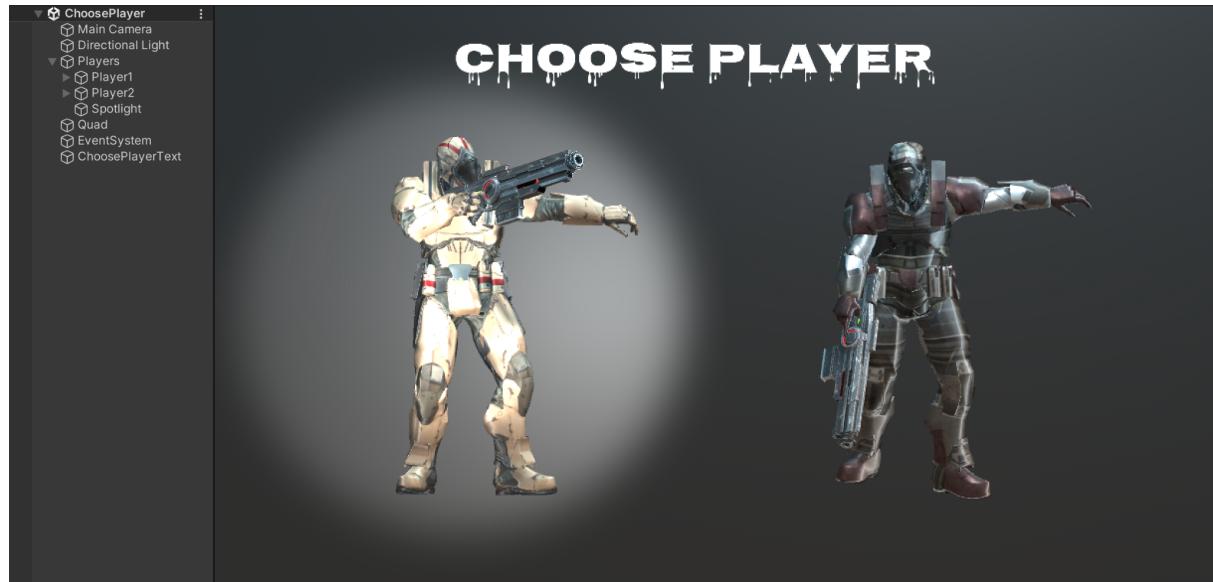
    public void Quit() => Application.Quit();
}
```

Le script vient activer ou non le menu lorsque nous en avons besoin et vient charger la scène 1 lorsque l'on appuie sur play. La scène 1 étant la scène de sélection du player.

Pour qu'un button soit fonctionnel, il faut lui passer une action “On Click()” comme ceci (pour le button Play par exemple):



Passons maintenant à la scène sélection du player :



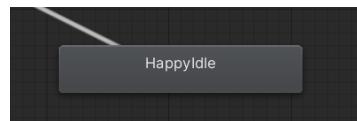
Pour réaliser cette scène, il a fallu 2 apparences de joueurs.

Il a fallu également :

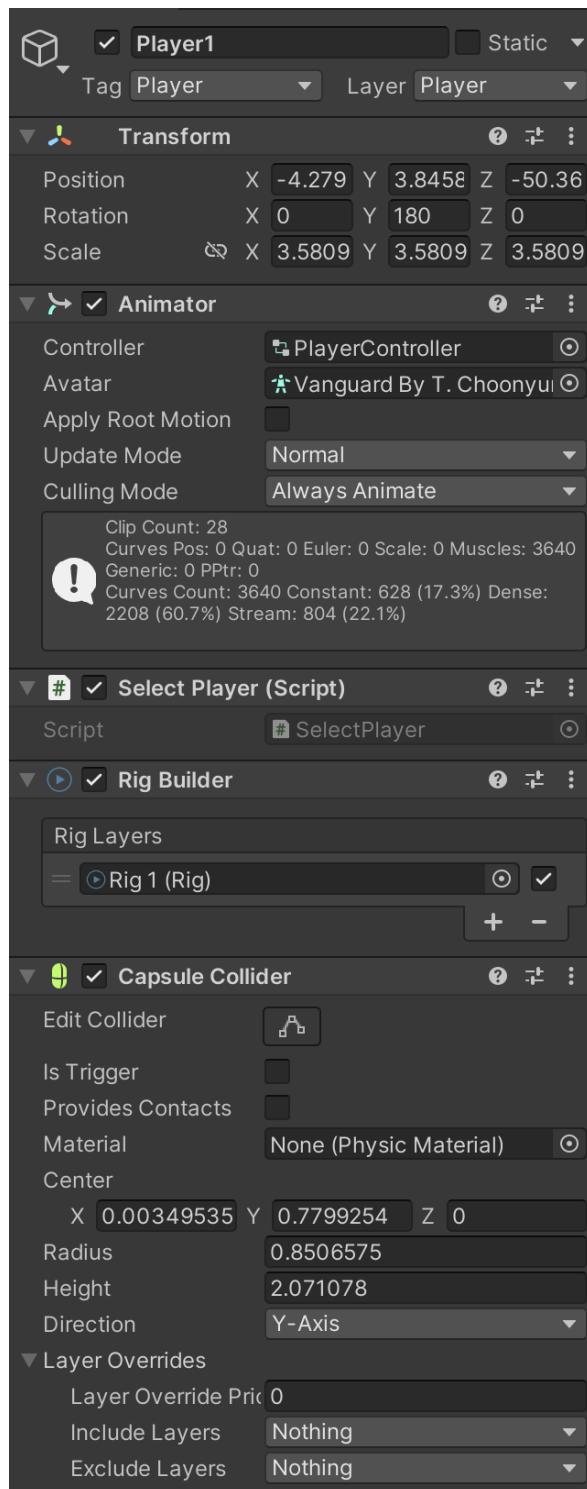
- un Quad qui est le fond noir de la scène,
- une direction light qui est la lumière lorsque l'on cible le personnage
- un texte : choose player.

Ces joueurs sont animés avec une animation “Idle”, donc statique qui permet d'animer le personnage en continu.

Pour cela, nous avons besoin d'un component animator dans lequel nous plaçons une animation :



Voilà une animation déposée dans un animator. Nous reviendrons en détail dans les animations du joueur lorsque nous aborderons la scène principale de notre jeu.  
Voici ici les components des player (ici player 1 par exemple)



Les 2 joueurs sont composés de la même manière :

- une animation "HappyIdle"
- un tag player afin de pouvoir appeler simplement l'objet
- un collider qui permettra de cliquer sur le joueur pour le choisir
- un script qui effectue la sélection du joueur que nous allons voir en détail
- un rig, qui permet de manipuler les articulations du player pour son positionnement.

```
public class SelectPlayer : MonoBehaviour
{
    5 références
    GameObject spotlight;

    0 références | Unity Message
    void Start()
    {
        spotlight = GameObject.Find("Spotlight").gameObject;
        Debug.Log(PlayerPrefs.GetString("Player"));
    }

    0 références | Unity Message
    void OnMouseEnter()
    {
        spotlight.GetComponent< AudioSource >().Play();
        spotlight.transform.position = new Vector3(transform.position.x, spotlight.transform.position.y, spotlight.transform.position.z);
    }

    0 références | Unity Message
    void OnMouseDown()
    {
        PlayerPrefs.SetString("Player", gameObject.name);
        SceneManager.LoadScene(2);
    }
}
```

Nous appelons tout d'abord l'objet lumière, qui se déplacera sur l'axe des x lorsque l'on cliquera sur un collider de joueur grâce à la méthode OnMouseEnter() qui est une méthode de base intégrée à unity.

Lorsque l'on relâche la souris (OnMouseDown()) on stockera le nom du joueur sélectionné dans PlayerPrefs qu'on retransmettra plus tard à la scène contenant notre map afin de savoir quel joueur nous avons sélectionné et donc, connaître lequel sera instancié dans la carte.

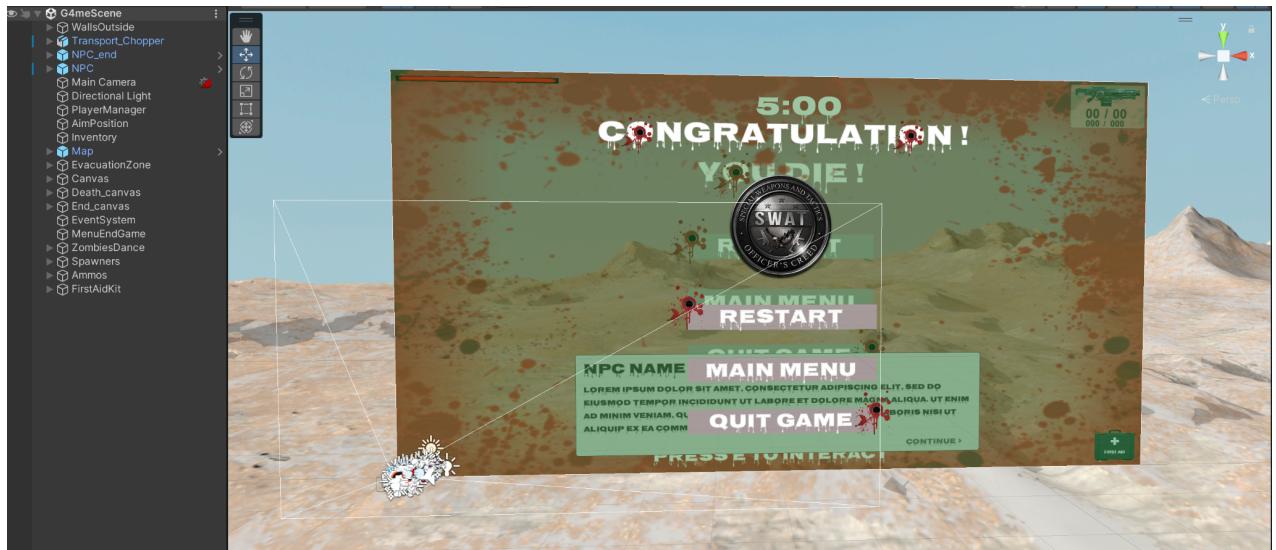
Un son est déclenché lorsque l'utilisateur survole un des joueur avec la souris grâce au component AudioSource

Le chargement de la scène 2 (scène de jeux) s'effectue lors du relâchement de la souris.

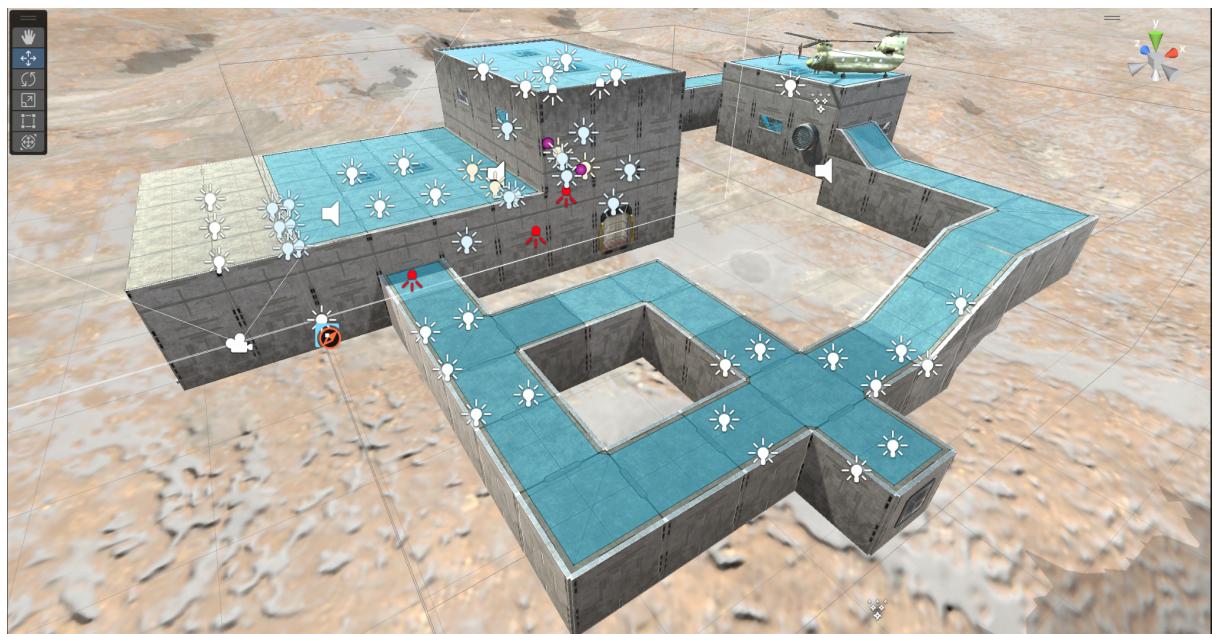
Nous allons maintenant entrer plus en détail dans la scène **G4meScene**.

Voici un aperçu de cette scène :

ici, les canvas contenant les menus et l'interface de jeu



ici la map de jeu, que nous verrons un peu plus en détail également.



Pour la transmission du joueur dans cette scène, comme expliqué précédemment le joueur est récupéré dans la variable **PlayerPrefs**.

Une fois le joueur récupérer celui ci est instancié et géré par l'objet **PlayerManager**

Celui ci contient le script **LoadPlayer** :

```
public class LoadPlayer : MonoBehaviour
{
    3 références
    public GameObject[] players;
    0 références | Unity Message
    void Awake()
    {
        string player = PlayerPrefs.GetString("Player");
        for (int i = 0; i < players.Length; i++)
        {
            if (player == players[i].name)
                Instantiate(players[i], transform.position, Quaternion.identity);
        }
    }
}
```

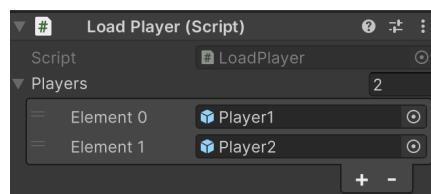
You, la semaine dernière • add player selection

Pour cela, nous mettons “**public GameObject[] players**” qui permettra de glisser déposer une liste d’objets (ici nos 2 joueurs prefabs) dans la variable players depuis l’interface de jeu.

**Awake** est une méthode de unity et est la première à être appelé (avant Start() qui se lance au démarrage du jeu). Cette méthode est très adaptée ici puisque nous voulons que notre joueur soit créé avant le jeu afin de ne pas avoir de bug qui nécessiterait la présence d’un joueur (comme le tracking des zombies par exemple qui recherchent un joueur à attaquer).

Ensuite, on va tout simplement regarder dans notre liste quel joueur à été sélectionné dans la scène d’avant grâce à son nom, et nous instancions son prefab. La méthode **Instantiate** est également une méthode intégré à unity qui permet de mettre dans la scène à une certaine position un prefab.

Voilà donc le rendu dans la scène :



Nous allons maintenant présenter les différentes actions possibles par le joueur.

Nous reviendrons dans cette partie sur les animations/l’animator.

Le joueur peut effectuer plusieurs déplacements :

- Marcher
- Courir
- Sauter
- S'accroupir

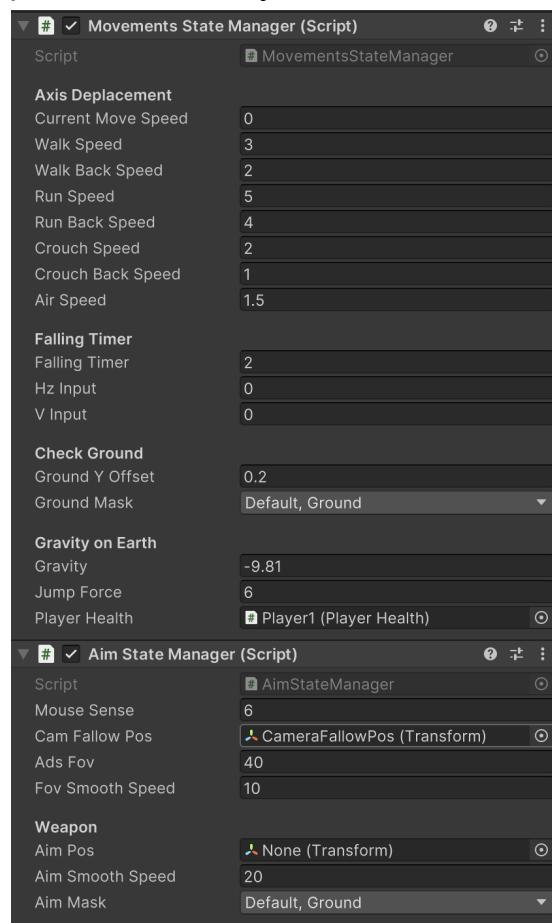
Il peut également effectuer des actions relatifs à son armement :

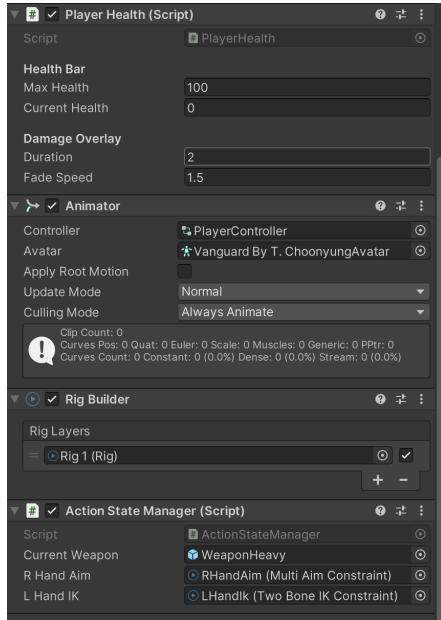
- Viser
- Tirer
- Recharger

Cela se fait en plusieurs étapes :

- Création du modèle (joueur, arme)
- Intégrations de scripts pour le joueur (mouvements) et pour l'arme (tir, recharge, bullet ..)
- Animations/animator et gestion des activations d'animations via des scripts

Voici les éléments composant le modèle joueur :





ainsi que les colliders et le tag player comme on l'a vu dans la sélection player.  
Voyons les scripts en détail :

```
[Header("State Movement")]
1 référence
public MovementBaseState previousState;
3 références
public MovementBaseState currentState;
2 références
public IdleState Idle = new IdleState();
0 références
public WalkState Walk = new WalkState();
0 références
public CrouchState Crouch = new CrouchState();
0 références
public RunState Run = new RunState();
0 références
public JumpState Jump = new JumpState();
```

Ici nous avons tous les états du joueurs afin de savoir dans quel état il se trouve : soit il marche, soit il court etc..

Nous allons voir par exemple comment fonctionne le JumpState, le fonctionnement est similaire pour les autres états :

```

0 références | You, il y a 2 mois | Author (You)
public class JumpState : MovementBaseState
{
    0 références
    public override void EnterState(MovementStateManager movement)
    {
        if (movement.previousState == movement.Idle)
            movement.anim.SetTrigger("IdleJump");
        else if ((movement.previousState == movement.Walk) || (movement.previousState == movement.Run))
            movement.anim.SetTrigger("RunJump");
    }

    0 références
    public override void UpdateState(MovementStateManager movement)
    {
        if (movement.jumped && movement.IsGrounded())
        {
            movement.jumped = false;
            //we want to keep the direction when we jump
            if ((movement.hzInput == 0) && (movement.vInput == 0))
                movement.SwitchState(movement.Idle);
            else if (Input.GetKey(KeyCode.LeftShift))
                movement.SwitchState(movement.Run);
            else
                movement.SwitchState(movement.Walk);
        }
    }
}

```

Dans ce code nous définissons donc son état d'entrée et mise à jour de son état en cours de jeu. Par exemple si on est sur place (movement.Idle) on effectue l'animation de saut sur place (Idle jump), sinon l'animation de "saut en courant" si nous marchons ou si nous courons (RunJump).  
Et donc on met à jour ensuite sa direction en saut afin de garder la même direction de déplacement en l'air. Pour cela, on garde donc le même état.

```

[Header("Axis Deplacement")]
1 référence
public float currentMoveSpeed;
0 références | 0 références
public float walkSpeed = 4, walkBackSpeed = 3;
0 références | 0 références
public float runSpeed = 7, runBackSpeed = 5;
0 références | 0 références
public float crouchSpeed = 3, crouchBackSpeed = 2;
1 référence
public float airSpeed = 1.5f;

```

Nous définissons les différentes vitesses de déplacements en fonction de son état.

```

[Header("Falling Timer")]
2 références
public float fallingTimer = 2f;
5 références
float currentFallingTimer;

2 références
[HideInInspector] public Vector3 dir;
4 références | 4 références
public float hzInput, vInput;
5 références
CharacterController controller;

[Header("Check Ground")]
1 référence
[SerializeField] float groundYOffset;
1 référence
[SerializeField] LayerMask groundMask;
3 références
Vector3 spherePos;

[Header("Gravity on Earth")]
1 référence
[SerializeField] float gravity = -9.81f;
1 référence
[SerializeField] float jumpForce = 5;
1 référence
[HideInInspector] public bool jumped;
5 références
Vector3 velocity;| You, il y a 2 mois

```

ici nous avons plusieurs variables pour définir la gravité lors de sa retombée (en saut), sa direction en fonction des axes, sa force de saut.

Le changement d'état se fait dans la fonction **SwitchState**

```

public void SwitchState(MovementBaseState state)
{
    currentState = state;
    currentState.EnterState(this);
}| You, il y a 2 mois • First commit with pla

```

```

void GetDirectionAndMove()
{
    hzInput = Input.GetAxis("Horizontal");
    vInput = Input.GetAxis("Vertical");
    Vector3 airDir = Vector3.zero;
    if (!IsGrounded())
        airDir = transform.forward * vInput + transform.right * hzInput;
    else
        dir = transform.forward * vInput + transform.right * hzInput;
    controller.Move((dir.normalized * currentMoveSpeed + airDir.normalized * airSpeed) * Time.deltaTime);
}

```

La méthode **GetDirectionAndMove** permet de savoir dans quelle direction le player se déplace et s'il est au sol ou en l'air et donc le déplacer avec la méthode **Move**, une autre méthode de unity qui prend en paramètre une direction \* un temps.

```

public bool IsGrounded()
{
    spherePos = new Vector3(transform.position.x, transform.position.y - groundYOffset, transform.position.z);
    if(Physics.CheckSphere(spherePos, controller.radius - 0.05f, groundMask))
    {
        if (currentFallingTimer <= 0)
        {
            playerHealth.TakeDamage(20);
        }
        currentFallingTimer = fallingTimer;
    }
    return true;
}

```

You, il y a 2 mois • First commit with player

Voici le détail de la méthode qui permet de savoir si on est au sol où non, grâce à une autre méthode de unity : CheckSphere qui va checker si elle touche un élément marqué comme "ground" ou "default" avec groundMask puisque notre joueur est représenté par une sphère.

Dans la scène :



Si le temps de chute dépasse une certaine hauteur (temps de chute), le joueur perd 20 points de vie.

Plusieurs méthodes permettent d'appliquer des forces et l'animation de falling lorsque le joueur chute:

```

1 référence
void Falling() => anim.SetBool("Falling", !IsGrounded() && !(previousState == Idle));

0 références
public void JumpForce() => velocity.y += jumpForce;

0 références
public void Jumped() => jumped = true;

```

Enfin la méthode **Gravity** permet d'appliquer une gravité au joueur

```
void Gravity()
{
    if (!IsGrounded())
    {
        velocity.y += gravity * Time.deltaTime;
        currentFallingTimer = currentFallingTimer - Time.deltaTime;
    }
    else if (velocity.y < 0) //fall
    {
        velocity.y = -2;
    }

    controller.Move(velocity * Time.deltaTime);
}
```

You, il y a 2 mois • First commit with player

On applique une force vers le bas sur l'axe y.

Maintenant, nous allons voir l'**aiming** du joueur :

```
void Start()
{
    aimPos = GameObject.Find("AimPosition").transform;
    vCam = GetComponentInChildren<CinemachineVirtualCamera>();
    hipFov = vCam.m_Lens.FieldOfView;
    anim = GetComponent<Animator>();
    SwitchState(hip);
}

0 références | Unity Message
void Update()
{
    xAxis += Input.GetAxisRaw("Mouse X") * mouseSense;
    yAxis -= Input.GetAxisRaw("Mouse Y") * mouseSense;
    yAxis = Mathf.Clamp(yAxis, -50, 50);

    vCam.m_Lens.FieldOfView = Mathf.Lerp(vCam.m_Lens.FieldOfView, currentFov, fovSmoothSpeed * Time.deltaTime);

    Vector2 screenCenter = new Vector2(Screen.width / 2, Screen.height / 2);
    Ray ray = Camera.main.ScreenPointToRay(screenCenter);

    if (Physics.Raycast(ray, out RaycastHit hit, Mathf.Infinity, aimMask))
        aimPos.position = Vector3.Lerp(aimPos.position, hit.point, aimSmoothSpeed * Time.deltaTime);

    currentState.UpdateState(this);
}

0 références | Unity Message
private void LateUpdate()
{
    //Mouse Cam fallow
    camFollowPos.localEulerAngles = new Vector3(yAxis, camFollowPos.localEulerAngles.y, camFollowPos.localEulerAngles.z);
    transform.eulerAngles = new Vector3(transform.eulerAngles.x, xAxis, transform.eulerAngles.z);
}
```

You, il y a 2 mois • First commit with player

Dans unity nous avons un objet AimPosition qui va donc déterminer la position de visée à suivre, une vCam qui est une caméra virtuelle attachée au player qui permet d'obtenir la vue à la 3ème personne..

Et donc dans ce script, nous allons venir mettre à jour son état, on définit le centre de l'écran avec screenCenter qui va donc être la position de visée (width/2, height/2) et on applique cette position sur l'objet AimPosition.

On met également à jour la caméra en fonction du déplacement de la souris avec une méthode de calcul mathématique intégré à unity : local EulerAngles. C'est une alternative aux quaternions que nous avons vu dans la méthode instantiate mais qui se base sur le même principe, des Vector3 qui définissent les 3 angles de rotations (x, y et z) puisque nous sommes en 3D.

Nous allons maintenant passer à l'action state management du player :

```
void Start()
{
    SwitchState(Default);
    ammo = currentWeapon.GetComponent<WeaponAmmo>();
    audioSource = currentWeapon.GetComponent< AudioSource >();
    anim = GetComponent< Animator >();
}

// Update is called once per frame
0 références | ⚡ Unity Message
void Update()
{
    currentState.UpdateState(this);
}

2 références
public void SwitchState(ActionBaseState state)
{
    currentState = state;
    currentState.EnterState(this);
}

0 références
public void WeaponReloaded()
{
    ammo.Reload();
    SwitchState(Default);
}

0 références
public void MagOut()
{
    audioSource.PlayOneShot(ammo.magOutSound);
}

0 références
public void MagIn()
{
    audioSource.PlayOneShot(ammo.magInSound);
}

0 références
public void ReleaseSlide()
{
    audioSource.PlayOneShot(ammo.releaseSlideSound);
}

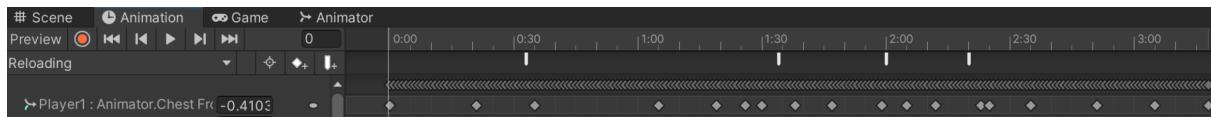
0 références
public void RefillAmmo()
{
    ammo.RefillAmmo();
}
```

Ici est gérée la partie rechargeement de l'arme du joueur ainsi que les sons associés au rechargeement. La recharge se fait en 3 temps :

- on enlève le chargeur
- on remet le chargeur
- on tire le levier d'armement

Un son différent sera joué en fonction de l'étape de rechargement qu'on effectue.

Dans l'animation reloading :



Les traits blancs représentent les actions effectuées à ce moment précis de l'animation (events). C'est ici qu'on appellera les méthodes : MagIn, MagOut et ReleaseSlide qui joueront les sons du rechargeement pendant l'animation.

Et enfin, pour notre joueur nous avons un dernier script : PlayerHealth, permettant de gérer la vie, les dégâts subis, la mort.

```
public void TakeHealth(int health)
{
    currentHealth += health;

    if (currentHealth > maxHealth)
        currentHealth = maxHealth;

    healthBar.SetHealth(currentHealth);
}

3 références
public void TakeDamage(int damage)
{
    currentHealth -= damage;
    healthBar.SetHealth(currentHealth);

    You, il y a 2 mois * add zombie, AI, anim/follow player, take damage t...
    if (currentHealth <= 0)
        Die();

    durationTimer = 0;
    overlay.color = new Color(overlay.color.r, overlay.color.g, overlay.color.b, 0.7f);
}

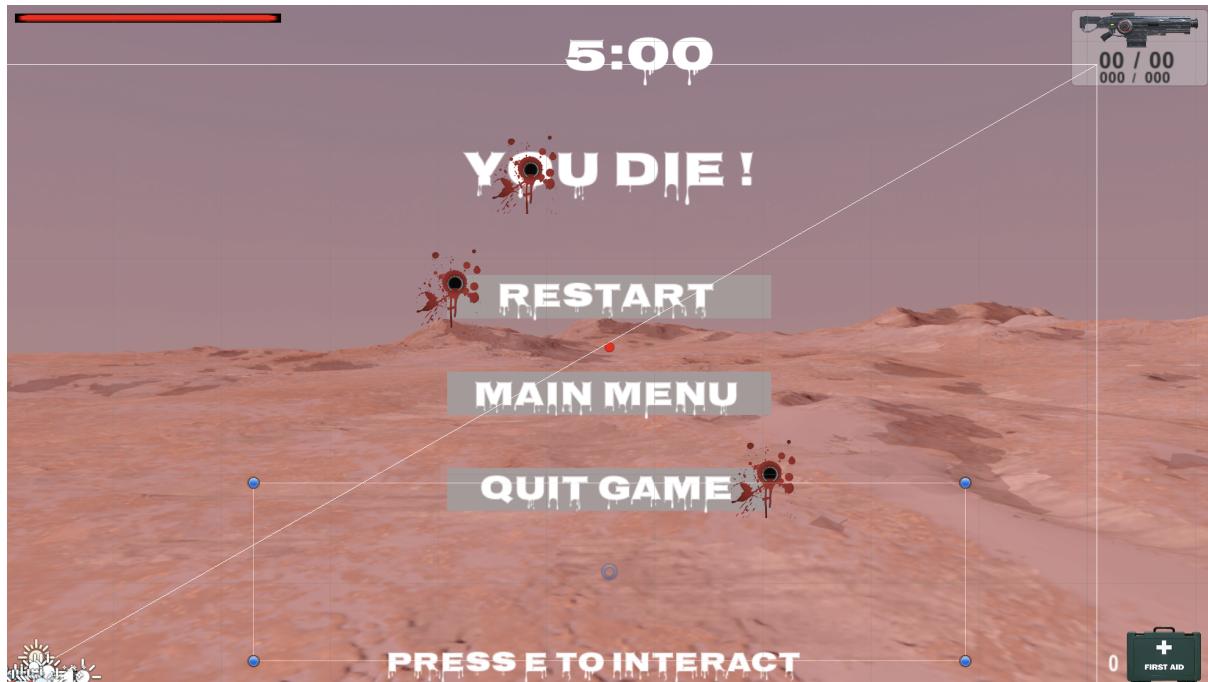
1 référence
void CheckHealth()
{
    if (currentHealth > 0)
        deathManager.SetActiveDeath(false);
    else
        Die();
}

2 références
public void Die()
{
    deathManager.SetActiveDeath(true);
    GetComponent<MovementsStateManager>().enabled = false;
    GetComponent<AimStateManager>().enabled = false;
    GetComponent<ActionStateManager>().enabled = false;
    timer.stopTimer = true;
}

1 référence
void UseFirstAid()
{
    Inventory inventory = Inventory.instance;
    if (inventory.health > 0)
    {
        TakeHealth(50);
        inventory.RemoveHealth(1);
    }
}
```

ici la méthode **UseFirstAidKit** qui permet de reprendre de la vie en utilisant la fonction **TakeHealth** grâce aux kits de vie que nous avons posés dans la map.

La méthode **TakeDamage** permet de perdre des points de vie. Si le nombre de points de vie se retrouve égal ou inférieur à 0 le joueur meurt et le jeu se termine.



Lorsque le joueur meurt, le menu de fin de jeu apparaît, le joueur ne peut plus se déplacer et le compteur est stoppé

Si le joueur parvient à rejoindre la zone d'évacuation sans mourir et dans le délai imparti le menu de fin de jeux est appelé



celui-ci s'affiche lorsque nous terminons le jeu et que le joueur passe la zone de fin :

```

void Start()
{
    magicRing = GameObject.FindGameObjectWithTag("EvacuateZone").GetComponent<MagicRing>();
    endManager = GameObject.Find("End_canvas").GetComponent<EndManager>();
}

0 références | Unity Message
void Update() => endManager.SetActiveEnd(isInside);

0 références | Unity Message
void OnTriggerEnter(Collider collision)
{
    if (collision.CompareTag("Player"))
        isInside = true;
    magicRing.toElevate = false;
}

0 références | Unity Message
void OnTriggerExit(Collider collision)
{
    if (collision.CompareTag("Player"))
        isInside = false;| You, il y a 5 jours • improve endgame
}

```

Voici les méthodes permettant la gestion de la victoire du jeu et de l'animation de fin de jeu

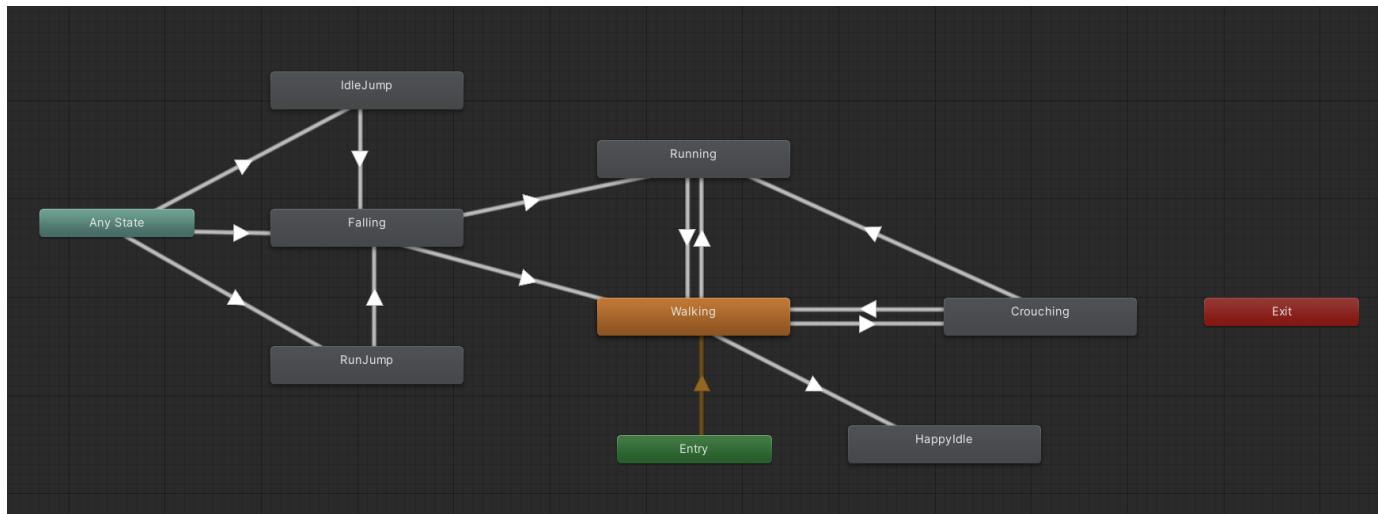
Lorsque le joueur rentre en collision avec le collider de l'EvacuateZone (zone matérialisé dans la carte par un cercle bleu/violet), le compteur est arrêté, le joueur figé, celui-ci est ensuite élevé vers le toit ou le menu de fin apparaît. Le menu de fin est déclenché lors d'une collision dans la méthode OnTriggerEnter.

## Animator

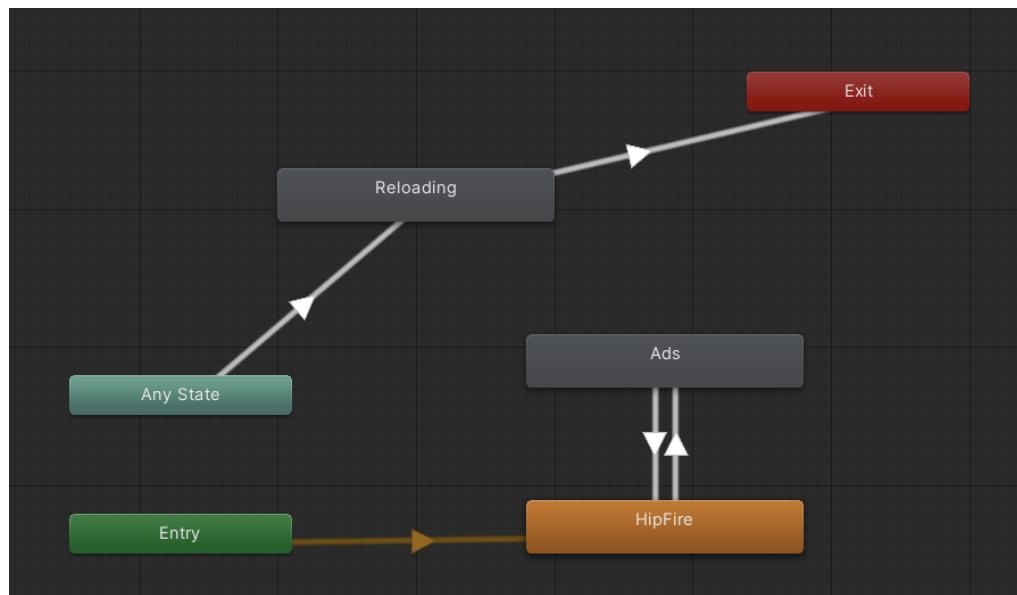
Dans cette partie nous allons voir en détail le fonctionnement de l'animator du player.

Il se décompose en 2 parties :

Ses actions de mouvements :

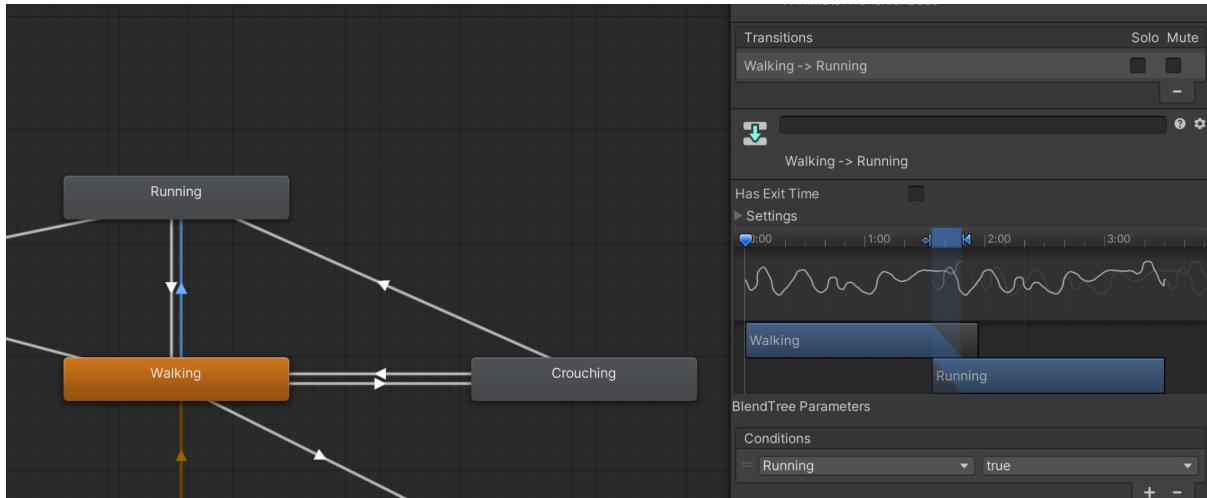


Ses actions de visée



Chaque bloc gris représente une animation. Celles-ci sont reliées par des flèches, se sont les conditions de transitions pour passer d'un état à l'autre, d'une animation à l'autre.

Par exemple :



ici on regarde de plus près la transition de la marche -> course. Il faut donc être dans la condition Running = true.

Et donc cela nous renvoie au code que nous avons vu sur les états avec notamment la première ligne d'EnterState :

```
public class RunState : MovementBaseState
{
    0 références
    public override void EnterState(MovementsStateManager movement)
    {
        movement.anim.SetBool("Running", true);
    }

    0 références
    public override void UpdateState(MovementsStateManager movement)
    {
        if (Input.GetKeyUp(KeyCode.LeftShift))
            ExitState(movement, movement.Walk);
        else if (movement.dir.magnitude < 0.01f)
            ExitState(movement, movement.Idle);

        if (movement.vInput < 0)
            movement.currentMoveSpeed = movement.runBackSpeed;
        else
            movement.currentMoveSpeed = movement.runSpeed;

        if (Input.GetKeyDown(KeyCode.Space))
        {
            movement.previousState = this;
            ExitState(movement, movement.Jump);
        }
    }

    3 références
    void ExitState(MovementsStateManager movement, MovementBaseState state)
    {
        movement.anim.SetBool("Running", false);
        movement.SwitchState(state);
    }
}
```

Voilà comment on applique cette condition d'animation.

Maintenant que nous avons fait le tour de tout cela, que nous reste il ?  
Pour le joueur il nous reste l'arme et notamment, les balles.  
Voici le script :

```
0 références | Unity Message
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.GetComponentInParent<EnemyHealth>())
    {
        EnemyHealth enemyHealth = collision.gameObject.GetComponentInParent<EnemyHealth>();
        enemyHealth.TakeDamage(weapon.damage);

        if (enemyHealth.health <= 0 && enemyHealth.isDead == false)
        {
            Rigidbody rb = collision.gameObject.GetComponent<Rigidbody>();
            rb.AddForce(dir * weapon.enemyKickbackForce, ForceMode.Impulse);
            enemyHealth.isDead = true;
        }
    }

    if (collision.gameObject.tag == "Impactable")
        Instantiate(ImpactsParticles, transform.position, transform.rotation);

    Destroy(this.gameObject);
}
```

On vient donc instancier un prefab de munition que nous avons créé, on lui applique une force (AddForce) qui est encore une méthode de unity qui va permettre à notre prefab de se déplacer en direction du centre où se trouve notre viseur.

Lorsque ce prefab rentre en collision avec un ennemi, et bien nous allons appeler le script de vie des ennemis (comme le player) afin qu'ils prennent des dégâts (TakeDamage).

Nous avons également mis des tags “Impactables” sur tous les éléments de la carte sur lesquels nous souhaitons voir apparaître des effets d’impacts de balles pour plus de réalisme.

Nous allons maintenant voir nos ennemis qui sont donc, des **zombies** !



TYPE A

TYPE B

TYPE C

Ils ont des animations/animator dans le même principe que le joueurs puisqu'ils se déplacent en direction du joueur et l'attaque quand ils sont à portée :

```
public class EnemyManager : MonoBehaviour
{
    4 références
    NavMeshAgent _agent;
    3 références
    Animator _animator;

    5 références
    private GameObject _target;
    2 références
    public GameObject _targetLook;
    1 référence
    public int damage = 20;
    0 références
    [HideInInspector] public float gravity = 9.81f;

    0 références | ⚙ Unity Message
    void Start()
    {
        _agent = GetComponent<NavMeshAgent>();
        _animator = GetComponent<Animator>();
        _target = GameObject.FindGameObjectWithTag("Player");
    }

    0 références | ⚙ Unity Message
    void Update()
    {
        float distanceToPlayer = Vector3.Distance(transform.position, _target.transform.position);
        _agent.destination = _target.transform.position;

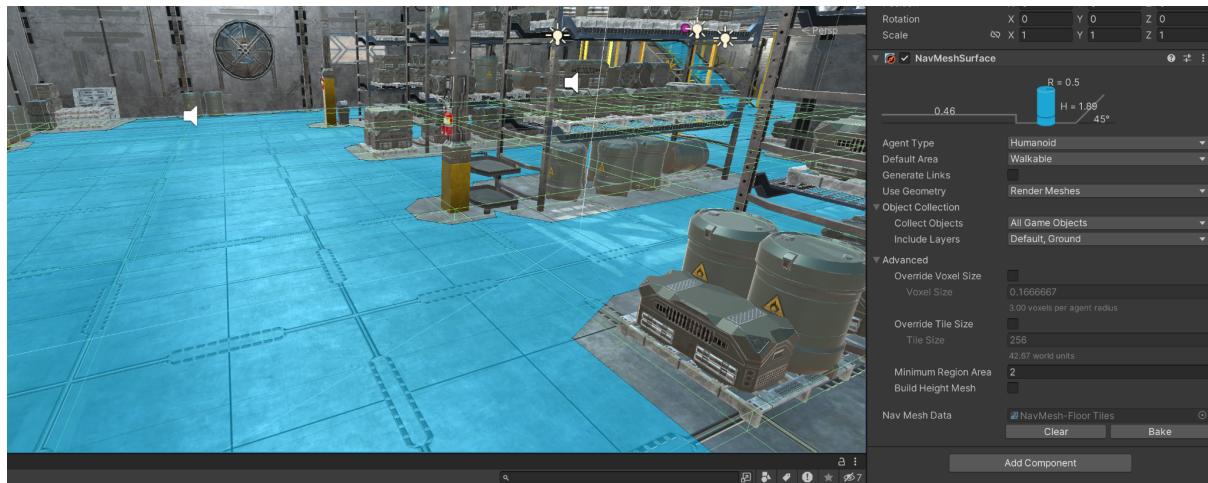
        if (_agent.velocity.magnitude <= 0.1f && distanceToPlayer <= _agent.stoppingDistance)
            _animator.SetBool("Walk", false);
        else
            _animator.SetBool("Walk", true);

        _targetLook.transform.LookAt(_target.transform);
        transform.rotation = Quaternion.Euler(0f, _targetLook.transform.eulerAngles.y, 0f);
    }

    0 références
    public void DamagePlayer() => _target.GetComponent<PlayerHealth>().TakeDamage(damage);
}
```

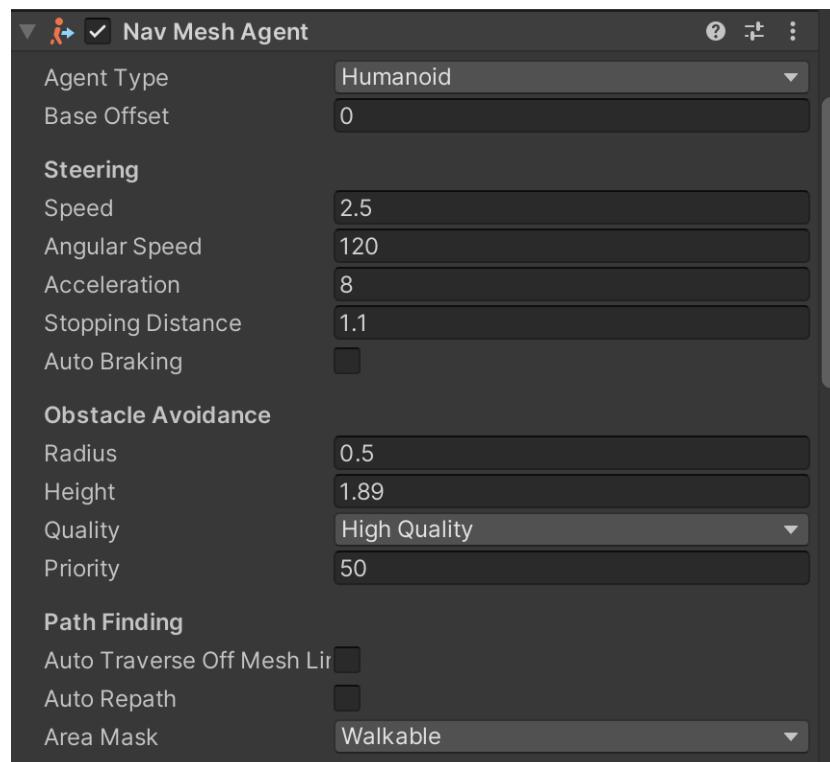
Ici on voit donc le target player dans la fonction start et l'importance d'instancier le player dans la méthode awake afin qu'il soit créé avant les zombies (Awake < Start).

Les zombies se déplacent sur une NavMeshSurface, c'est un système d'IA qui va calculer la surface au sol "walkable", marchable et définir un chemin bleuté sur lequel les zombies pourront se déplacer. Pour cela, quelques paramètres à insérer comme la hauteur minimale, sur quels types d'objets on l'applique etc. Voici le rendu en jeu :



Donc le zombie marche tant qu'il n'arrive pas à distance du player, quand il y est, il s'arrête et attaque le joueur. On retrouve le principe de quaternion et d'euler car le zombie nous "pourchasse", c'est à dire qu'il se dirige selon les 3 axes x, y, et z afin de suivre les mouvements du joueurs dans ces 3 dimensions.

Pour cela, le zombie doit être un `NavMeshAgent`. On retrouve également quelques caractéristiques paramétrables pour le zombie comme sa vitesse, sa distance d'attaque (traduit par stopping distance), sa hauteur (pour qu'il ne puisse pas passer à travers des espaces où le plafond est trop bas par exemple) :



Nous avons appliqué un système de ragdoll sur les zombies pour plus de fun !

```
public class RagdollManager : MonoBehaviour
{
    3 références
    Rigidbody[] rbs;

    0 références | Unity Message
    void Start()
    {
        rbs = GetComponentsInChildren<Rigidbody>();
        foreach(Rigidbody rb in rbs)
            rb.isKinematic = true;
    }

    0 références
    public void TriggerRagdoll()
    {
        foreach(Rigidbody rb in rbs)
            rb.isKinematic = false;
    }
}
```

Tous les membres du zombie sont stockés dans rbs et la rigidité sur les membres est retirée.

Les zombies apparaissent par spawn que nous avons dupliqués à plusieurs endroits de la scène. Un certain nombres de paramètres peuvent être renseignés afin que les spawns ne se ressemblent pas comme le nombre de types de zombies ou encore le temps de spawn de départ et l'intervalle de temps entre chaque zombie qui spawn.

Voici le script :

```
void Start()
{
    nbTotalSpawn = nbZombieTypeA + nbZombieTypeB + nbZombieTypeC;
    player = GameObject.FindGameObjectWithTag("Player");
    StartCoroutine(spawnZomby(spawnInterval, zombieTypeA, zombieTypeB, zombieTypeC));
}

2 références
private IEnumerator spawnZomby(float interval, GameObject zombieA, GameObject zombieB, GameObject zombieC)
{
    float distance = Vector3.Distance(player.transform.position, transform.position);
    if (distance < minDistance)
    {
        yield return new WaitForSeconds(spawnStart);

        for (int i = 0; i < nbTotalSpawn; i++)
        {
            spawnCount++;
            if (nbZombieTypeA > 0)
            {
                InstantiateZombieWithRandomPosition(zombieA, interval);
                nbZombieTypeA--;
                yield return new WaitForSeconds(interval);
            }

            if (nbZombieTypeB > 0)
            {
                InstantiateZombieWithRandomPosition(zombieB, interval);
                nbZombieTypeB--;
                yield return new WaitForSeconds(interval);
            }

            if (nbZombieTypeC > 0)
            {
                InstantiateZombieWithRandomPosition(zombieC, interval);
                nbZombieTypeC--;
                yield return new WaitForSeconds(interval);
            }
        }
    }

    yield return new WaitForSeconds(1);

    if (spawnCount < 1)
        StartCoroutine(spawnZomby(interval, zombieA, zombieB, zombieC));
}
}

3 références
private void InstantiateZombieWithRandomPosition(GameObject zombie, float interval)
{
    Vector3 randomPosition = transform.position;
    randomPosition.x = randomPosition.x + Random.Range(-randomInterval, randomInterval);
    Instantiate(zombie, randomPosition, Quaternion.identity);
}
```

Lorsque l'on parle de délai dans unity, il y a 2/3 choses qui sont associés :

- IEnumerator
- StartCoroutine
- Instantiate

Instantiate on en a déjà parlé, cela permet de déposer un prefab dans la scène à une position donnée.

IEnumerator est un type de retour d'une méthode et est obligatoire si l'on souhaite appeler la méthode de unity : StartCoroutine qui va créer une nouvelle "task" en parallèle de la main task. C'est ce qui nous permet donc d'attendre un certain temps avant d'effectuer une action.

Le type `IEnumerator` exige également un retour de la sorte : `yield return ...` ce qui permet donc, dans notre code, d'appeler `WaitForSeconds` car nous souhaitons faire attendre l'apparition du prochain zombie afin d'avoir un délai réaliste entre 2 zombies qui spawns et ne pas les voir tous apparaître d'un coup.

Pour survivre à nos zombies, nous avons un système d'inventaire et un système de ramassage d'objets :

inventory :

```
public void Start()
{
    currentWeapon = GameObject.FindGameObjectWithTag("Weapon");
    ammo = currentWeapon.GetComponent<WeaponAmmo>();
}

0 références | Unity Message
private void Awake()
{
    if(instance != null)
        return;

    instance = this;
}

0 références
public void AddHealth(int count)
{
    health += count;
    FirstAidText.text = health.ToString();
}

0 références
public void RemoveHealth(int count)
{
    health -= count;
    FirstAidText.text = health.ToString();
}

0 références
public void RefillAmmo() => ammo.RefillAmmo();
```

Dans la méthode `Awake`, nous faisons ce qu'on appelle un : singleton. C'est une manière de rendre unique une instance de la classe. C'est-à-dire que si l'objet est créé en double, il y aura un problème et on s'assure ainsi que chaque élément de notre inventaire est unique (même si c'est des objets de même type !).

C'est donc ici que l'on crée les méthodes qui vont appliquer add ou remove à la vie, ainsi que la recharge de munition.

ramassage :

```
0 références | Unity Message
void Start() => weaponAmmo = GameObject.FindGameObjectWithTag("Weapon").GetComponent<WeaponAmmo>();

0 références | Unity Message
private void OnTriggerEnter(Collider collision)
{
    if (collision.CompareTag("Player"))
    {
        if(type == "AMMO" && (weaponAmmo.currentAmmo != weaponAmmo.clipSize || weaponAmmo.extraAmmo != weaponAmmo.maxExtraAmmo))
        {
            AudioSource.PlayClipAtPoint(audioClip, transform.position);
            Inventory.instance.RefillAmmo();
            ObjectManager();
        }

        if (type == "HEALTH")
        {
            AudioSource.PlayClipAtPoint(audioClip, transform.position);
            Inventory.instance.AddHealth(1);
            ObjectManager();
        }
    }
}

2 références
private void ObjectManager()
{
    if (respawn)
    {
        gameObject.SetActive(false);
        Invoke(nameof(Respawn), delayRespawn);
    }
    else
        Destroy(gameObject);
}

1 référence
private void Respawn() => gameObject.SetActive(true);
```

Et donc, c'est ici que nous allons appliquer les effets lorsque nous les ramassons. Si on est pleins en munitions il ne se passe rien, sinon on recharge complètement toutes nos munitions.

Pour la vie, on stock dans l'inventaire un kit de soin qu'on pourra utiliser lorsqu'on aura perdu de la vie en appuyant sur h.

On a également instauré un temps de respawn sur les objets.

Il nous reste maintenant un dernier élément principal de notre jeu, le **pnj** : il a une animation sur place histoire de le rendre un peu plus “vivant” et un script qui va nous permettre d'interagir avec lui, d'afficher un panel de discussion et d'appuyer sur suivant pour voir ces instructions :



On remarque également que sur ces 2 images on aperçoit un kit de vie et un kit de munition.

```

private void Awake()
{
    if (instance != null)
        return;

    instance = this;
    sentences = new Queue<string>();
}

0 références
public void StartDialogue(DialogueNPC dialogue)
{
    animator.SetBool("isOpen", true);

    nameText.text = dialogue.name;
    sentences.Clear();

    foreach (string sentence in dialogue.sentences)
        sentences.Enqueue(sentence);

    DisplayNextSentence();
}

1 référence
public void DisplayNextSentence()
{
    if (sentences.Count == 0)
    {
        EndDialogue();
        return;
    }

    //get next sentence
    string sentence = sentences.Dequeue();
    StopAllCoroutines();
    StartCoroutine(TypeSentence(sentence));
}

1 référence
IEnumerator TypeSentence(string sentence)
{
    dialogueText.text = "";
    foreach (char letter in sentence.ToCharArray())
    {
        dialogueText.text += letter;
        yield return new WaitForSeconds(0.05f);
    }
}

1 référence
void EndDialogue() => animator.SetBool("isOpen", false);

```

Le panel dialogue a une petite animation de montée descente (afin de le voir et ne plus le voir).

Puis on vient écrire les phrases que l'on souhaite lui faire dire dans des strings et enfin on les fait défiler comme une machine à écrire à l'aide de l'IEnumerator et StartCoroutine vu un peu avant ainsi qu'une méthode de unity "sentence.ToCharArray()" qui vient récupérer chaque caractère du text. On attend un temps très court entre chaque caractère ce qui fait un système de lecture temps réel et plus fluide pour le lecteur.

```

public class DialogueTrigger : MonoBehaviour
{
    4 références
    private Text interactUI;
    1 référence
    public DialogueNPC dialogue;

    3 références
    public bool isInRange;

    0 références | ⚙ Unity Message
    private void Awake()
    {
        interactUI = GameObject.FindGameObjectWithTag("InteractUI").GetComponent<Text>();
        interactUI.enabled = false;
    }

    0 références | ⚙ Unity Message
    void Update()
    {
        if (isInRange && Input.GetKeyDown(KeyCode.E))
        {
            TriggerDialogue();
        }
    }

    0 références | ⚙ Unity Message
    private void OnTriggerEnter(Collider collision)
    {
        if (collision.CompareTag("Player"))
        {
            isInRange = true;
            interactUI.enabled = true;
        }
    }

    0 références | ⚙ Unity Message
    private void OnTriggerExit(Collider collision)
    {
        if (collision.CompareTag("Player"))
        {
            isInRange = false;
            interactUI.enabled = false;
        }
    }

    1 référence
    void TriggerDialogue() => DialogueManager.instance.StartDialogue(dialogue);
}

```

Et c'est donc dans ce code que l'on vient interagir avec le pnj en appuyant sur E lorsque l'on rentre dans un collider situé autour du pnj.

Il ne faut pas oublier de cocher l'option “IsTrigger” d'un collider si on souhaite rentrer dans ce collider et interagir avec.

C'est exactement le même principe pour déclencher le timer en début de partie en passant la porte, ou pour détecter l'ouverture de porte quand on s'en approche etc

...

Allé, je vous le montre aussi :

```
0 références | Unity Message
void Update()
{
    if (opening)
        OpenDoor();
    if (closing)
        CloseDoor();
}

0 références | Unity Message
void OnTriggerEnter(Collider collision)
{
    if (collision.CompareTag("Player"))
    {
        opening = true;
        closing = false;
    }
}

0 références | Unity Message
void OnTriggerExit(Collider collision)
{
    if (collision.CompareTag("Player"))
    {
        opening = false;
        closing = true;
    }
}

1 référence
void OpenDoor()
{
    float movement = speed * Time.deltaTime;
    currentValue += movement;
    if (currentValue <= maxOpenValue)
        door.position = new Vector3(door.position.x, door.position.y + movement, door.position.z);
    else
        opening = false;
}

1 référence
void CloseDoor()
{
    float movement = speed * Time.deltaTime;
    currentValue -= movement;
    if (currentValue >= 0)
        door.position = new Vector3(door.position.x, door.position.y - movement, door.position.z);
    else
        closing = false;
}
```

Donc si le joueur rentre en collision avec un collider que nous avons mis de part et d'autre de la porte, celle-ci s'ouvre automatiquement quand on y rentre, et se ferme automatiquement quand on en sort. Pour cela, on a joué avec la position y de la porte (door.position.y +/- movement).